

Bounded Model Checking for Probabilistic Programs^{*}

Nils Jansen², Christian Dehnert¹, Benjamin Lucien Kaminski¹,
Joost-Pieter Katoen¹, and Lukas Westhofen¹

¹ RWTH Aachen University, Germany

² University of Texas at Austin, USA

Abstract. In this paper we investigate the applicability of standard model checking approaches to verifying properties in probabilistic programming. As the operational model for a standard probabilistic program is a potentially infinite parametric Markov decision process, no direct adaptation of existing techniques is possible. Therefore, we propose an on-the-fly approach where the operational model is successively created and verified via a step-wise execution of the program. This approach enables to take key features of many probabilistic programs into account: non-determinism and conditioning. We discuss the restrictions and demonstrate the scalability on several benchmarks.

1 Introduction

Probabilistic programs are imperative programs, written in languages like **C**, **Scala**, **Prolog**, or **ML**, with two added constructs: (1) the ability to draw values at random from probability distributions, and (2) the ability to condition values of variables in a program through observations. In the past years, such programming languages became very popular due to their wide applicability for several different research areas [1]: Probabilistic programming is at the heart of *machine learning* for describing distribution functions; *Bayesian inference* is pivotal in their analysis. They are central in *security* for describing cryptographic constructions (such as randomized encryption) and security experiments. In addition, probabilistic programs are an active research topic in *quantitative information flow*. Moreover, *quantum programs* are inherently probabilistic due to the random outcomes of quantum measurements. All in all, the simple and intuitive syntax of probabilistic programs makes these different research areas accessible to a broad audience.

However, although these programs typically consist of a few lines of code, they are often hard to understand and analyze; bugs, for instance *non-termination* of a program, can easily occur. It seems of utmost importance to be able to automatically prove properties like “*Is the probability for termination of the program*

^{*} This work has been partly funded by the awards AFRL # FA9453-15-1-0317, ARO # W911NF-15-1-0592 and ONR # N00014-15-IP-00052 and is supported by the Excellence Initiative of the German federal and state government.

at least 90%” or “Is the expected value of a certain program variable at least 5 after successful termination?”. Approaches based on the simulation of a program to show properties or infer probabilities have been made in the past [2,3]. However, to the best of our knowledge there is no work which exploits well-established *model checking algorithms* for probabilistic systems such as Markov decision processes (MDP) or Markov chains (MCs), as already argued to be an interesting avenue for the future in [1].

As the operational semantics for a probabilistic program can be expressed as a (possible infinite) MDP [4], it seems worthwhile to investigate the opportunities there. However, probabilistic model checkers like PRISM [5], *iscasMc* [6], or MRMC [7] offer efficient methods only for *finite models*.

We make use of the simple fact that for a finite unrolling of a program the corresponding operational MDP is also finite. Starting from a profound understanding of the (intricate) probabilistic program semantics—including features such as observations, unbounded (and hence possibly diverging) loops, and nondeterminism—we show that with each unrolling of the program both conditional reachability probabilities and conditional expected values of program variables increase monotonically. This gives rise to a *bounded model-checking approach* for verifying probabilistic programs. This enables for a user to write a program and automatically verify it against a desired property without further knowledge of the programs semantics.

We extend this methodology to the even more complicated case of *parametric probabilistic programs*, where probabilities are given by functions over parameters. At each iteration of the bounded model checking procedure, parameter valuations violating certain properties are guaranteed to induce violation at each further iteration.

We demonstrate the applicability of our approach using five well-known benchmarks from the literature. Using efficient model building and verification methods, our prototype is able to prove properties where either the state space of the operational model is infinite or consists of millions of states.

Related Work. Besides the tools employing probabilistic model checking as listed above, one should mention the approach in [8], where *finite abstractions* of the operational semantics of a program were verified. However, this was defined for programs without parametric probabilities or observe statements. In [9], verification on partial operational semantics is theoretically discussed for termination probabilities.

The paper is organized as follows: In Section 2, we introduce the probabilistic models we use, the probabilistic programming language, and the structured operational semantics (SOS) rules to construct an operational (parametric) MDP. Section 3 first introduces formal concepts needed for the finite unrollings of the program, then shows how expectations and probabilities grow monotonically, and finally explains how this is utilized for bounded model checking. In Section 4, an extensive description of used benchmarks, properties and experiments is given before the paper concludes with Section 5.

2 Preliminaries

2.1 Distributions and Polynomials

A *probability distribution* over a finite or countably infinite set X is a function $\mu: X \rightarrow [0, 1] \subseteq \mathbb{R}$ with $\sum_{x \in X} \mu(x) = 1$. The set of all distributions on X is denoted by $\text{Distr}(X)$. Let V be a finite set of *parameters* over \mathbb{R} . A *valuation* for V is a function $u: V \rightarrow \mathbb{R}$. Let $\mathbb{Q}[V]$ denote the set of multivariate *polynomials* with rational coefficients and \mathbb{Q}_V the set of *rational functions* (fractions of polynomials) over V . For $g \in \mathbb{Q}[V]$ or $g \in \mathbb{Q}_V$, let $g[u]$ denote the evaluation of g at u . We write $g = 0$ if g can be reduced to 0, and $g \neq 0$ otherwise.

2.2 Probabilistic Models

First, we introduce parametric probabilistic models which can be seen as transition systems where the transitions are labelled with polynomials in $\mathbb{Q}[V]$.

Definition 1 (pMDP and pMC). A parametric Markov decision process (pMDP) is a tuple $\mathcal{M} = (S, s_I, \text{Act}, \mathcal{P})$ with a countable set S of states, an initial state $s_I \in S$, a finite set Act of actions, and a transition function $\mathcal{P}: S \times \text{Act} \times S \rightarrow \mathbb{Q}[V]$ satisfying for all $s \in S$: $\text{Act}(s) \neq \emptyset$, where V is a finite set of parameters over \mathbb{R} and $\text{Act}(s) = \{\alpha \in \text{Act} \mid \exists s' \in S. \mathcal{P}(s, \alpha, s') \neq 0\}$. If for all $s \in S$ it holds that $|\text{Act}(s)| = 1$, \mathcal{M} is called a parametric discrete-time Markov chain (pMC), denoted by \mathcal{D} .

At each state, an action is chosen *nondeterministically*, then the successor states are determined *probabilistically* as defined by the transition function. $\text{Act}(s)$ is the set of *enabled* actions at state s . As $\text{Act}(s)$ is non-empty for all $s \in S$, there are no deadlock states. For pMCs there is only one single action per state and we write the transition probability function as $\mathcal{P}: S \times S \rightarrow \mathbb{Q}[V]$, omitting that action. *Rewards* are defined using a *reward function* $\text{rew}: S \rightarrow \mathbb{R}$ which assigns rewards to states of the model. Intuitively, the reward $\text{rew}(s)$ is earned upon *leaving* the state s .

Schedulers. The nondeterministic choices of actions in pMDPs can be resolved using *schedulers*³. In our setting it suffices to consider memoryless deterministic schedulers [10]. For more general definitions we refer to [11].

Definition 2. (Scheduler) A scheduler for pMDP $\mathcal{M} = (S, s_I, \text{Act}, \mathcal{P})$ is a function $\mathfrak{S}: S \rightarrow \text{Act}$ with $\mathfrak{S}(s) \in \text{Act}(s)$ for all $s \in S$.

Let $\text{Sched}^{\mathcal{M}}$ denote the set of all schedulers for \mathcal{M} . Applying a scheduler to a pMDP yields an *induced parametric Markov chain*, as all nondeterminism is resolved, i.e., the transition probabilities are obtained w.r.t. the choice of actions.

Definition 3. (Induced pMC) Given a pMDP $\mathcal{M} = (S, s_I, \text{Act}, \mathcal{P})$, the pMC induced by $\mathfrak{S} \in \text{Sched}^{\mathcal{M}}$ is given by $\mathcal{M}^{\mathfrak{S}} = (S, s_I, \text{Act}, \mathcal{P}^{\mathfrak{S}})$, where

$$\mathcal{P}^{\mathfrak{S}}(s, s') = \mathcal{P}(s, \mathfrak{S}(s), s'), \quad \text{for all } s, s' \in S.$$

³ Also referred to as adversaries, strategies, or policies.

Valuations. Applying a *valuation* u to a pMDP \mathcal{M} , denoted $\mathcal{M}[u]$, replaces each polynomial g in \mathcal{M} by $g[u]$. We call $\mathcal{M}[u]$ the *instantiation* of \mathcal{M} at u . A valuation u is *well-defined* for \mathcal{M} if the replacement yields *probability distributions* at all states; the resulting model $\mathcal{M}[u]$ is a Markov decision process (MDP) or, in absence of nondeterminism, a Markov chain (MC).

Properties. For our purpose we consider *conditional reachability properties* and *conditional expected reward properties* in MCs. For more detailed definitions we refer to [11, Ch. 10]. Given an MC \mathcal{D} with state space S and initial state s_I , let $\Pr^{\mathcal{D}}(\neg\Diamond U)$ denote the probability *not* to reach a set of undesired states U from the initial state s_I within \mathcal{D} . Furthermore, let $\Pr^{\mathcal{D}}(\Diamond T \mid \neg\Diamond U)$ denote the conditional probability to reach a set of target states $T \subseteq S$ from the initial state s_I within \mathcal{D} , given that no state in the set U is reached. We use the standard probability measure on infinite paths through an MC. For threshold $\lambda \in [0, 1] \subseteq \mathbb{R}$, the reachability property, asserting that a target state is to be reached with conditional probability at most λ , is denoted $\varphi = \mathbb{P}_{\leq \lambda}(\Diamond T \mid \neg\Diamond U)$. The property is satisfied by \mathcal{D} , written $\mathcal{D} \models \varphi$, iff $\Pr^{\mathcal{D}}(\Diamond T \mid \neg\Diamond U) \leq \lambda$. This is analogous for comparisons like $<$, $>$, and \geq .

The reward of a path through an MC \mathcal{D} until T is the sum of the rewards of the states visited along on the path before reaching T . The expected reward of a finite path is given by its probability times its reward. Given $\Pr^{\mathcal{D}}(\Diamond T) = 1$, the conditional expected reward of reaching $T \subseteq S$, given that no state in set $U \subseteq S$ is reached, denoted $\text{ER}^{\mathcal{D}}(\Diamond T \mid \neg\Diamond U)$, is the expected reward of all paths accumulated until hitting T while not visiting a state in U in between divided by the probability of not reaching a state in U (i.e., divided by $\Pr^{\mathcal{D}}(\neg\Diamond U)$). An expected reward property is given by $\psi = \mathbb{E}_{\leq \kappa}(\Diamond T \mid \neg\Diamond U)$ with threshold $\kappa \in \mathbb{R}_{\geq 0}$. The property is satisfied by \mathcal{D} , written $\mathcal{D} \models \psi$, iff $\text{ER}^{\mathcal{D}}(\Diamond T \mid \neg\Diamond U) \leq \kappa$. Again, this is analogous for comparisons like $<$, $>$, and \geq . For details about conditional probabilities and expected rewards see [12].

Reachability probabilities and expected rewards for MDPs are defined on induced MCs for specific schedulers. We take here the conservative view that a property for an MDP has to hold for *all possible schedulers*.

Parameter Synthesis. For pMCs, one is interested in *synthesizing* well-defined valuations that induce satisfaction or violation of the given specifications [13]. In detail, for a pMC \mathcal{D} , a rational function $g \in \mathbb{Q}_V$ is computed which—when instantiated by a well-defined valuation u for \mathcal{D} —evaluates to the actual reachability probability or expected reward for \mathcal{D} , i.e., $g[u] = \Pr^{\mathcal{D}[u]}(\Diamond T)$ or $g[u] = \text{ER}^{\mathcal{D}[u]}(\Diamond T)$. For pMDPs, schedulers inducing *maximal* or *minimal* probability or expected reward have to be considered [14].

2.3 Conditional Probabilistic Guarded Command Language

We first present a programming language which is an extension of Dijkstra’s guarded command language [15] with a binary probabilistic choice operator, yielding the *probabilistic guarded command language* (pGCL) [16]. In [17], pGCL

was endowed with *observe statements*, giving rise to conditioning. The syntax of this *conditional probabilistic guarded command language* (cpGCL) is given by

$$\begin{aligned} \mathcal{P} ::= & \text{skip} \mid \text{abort} \mid x := E \mid \mathcal{P}; \mathcal{P} \mid \text{if } G \text{ then } \mathcal{P} \text{ else } \mathcal{P} \\ & \mid \{\mathcal{P}\} [g] \{\mathcal{P}\} \mid \{\mathcal{P}\} \square \{\mathcal{P}\} \mid \text{while } (G) \{\mathcal{P}\} \mid \text{observe } (G) \end{aligned}$$

Here, x belongs to the set of *program variables* \mathcal{V} ; E is an arithmetical expression over \mathcal{V} ; G is a *Boolean expression* over arithmetical expressions over \mathcal{V} . The *probability* is given by a polynomial $g \in \mathbb{Q}[V]$. Most of the cpGCL instructions are self-explanatory; we elaborate only on the following: For cpGCL-programs P and Q , $\{P\} [g] \{Q\}$ is a *probabilistic choice* where P is executed with probability g and Q with probability $1-g$; analogously, $\{P\} \square \{Q\}$ is a *nondeterministic choice* between P and Q ; **abort** is syntactic sugar for the diverging program **while** (true) {skip}. The statement **observe** (G) for the Boolean expression G *blocks* all program executions violating G and induces a *rescaling* of probability of the remaining execution traces so that they sum up to one. For a cpGCL-program P , the set of *program states* is given by $\mathbb{S} = \{\sigma \mid \sigma: \mathcal{V} \rightarrow \mathbb{Q}\}$, i.e., the set of all variable valuations. We assume all variables to be assigned zero prior to execution or at the start of the program. This initial variable valuation $\sigma_I \in \mathbb{S}$ with $\forall x \in \mathcal{V}. \sigma_I(x) = 0$ is called the *initial state* of the program.

Example 1. Consider the following cpGCL-program with variables x and c :

```

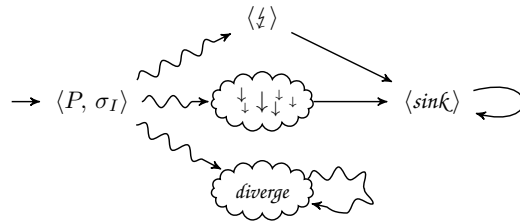
1 while (c = 0) {
2     { x := x + 1 } [0.5] { c := 1 }
3 };
4 observe "x is odd"

```

While c is 0, the loop body is iterated: With probability $1/2$ either x is incremented by one or c is set to one. After leaving the loop, the event that the valuation of x is odd is *observed*, which means that all program executions where x is even are blocked. Properties of interest for this program would, e.g., concern the termination probability, or the expected value of x after termination. \triangle

2.4 Operational Semantics for Probabilistic Programs

We now introduce an operational semantics for cpGCL-programs which is given by an MDP as in Definition 1. The structure of such an operational MDP is schematically depicted below.



Squiggly arrows indicate reaching certain states via possibly multiple paths and states; the clouds indicate that there might be several states of the particular kind. $\langle P, \sigma_I \rangle$ marks the initial state of the program P . In general the states of the operational MDP are of the form $\langle P', \sigma' \rangle$ where P' is the program that is left to be executed and σ' is the current variable valuation.

All runs of the program (paths through the MDP) are either *terminating* and eventually end up in the $\langle \text{sink} \rangle$ state, or are *diverging* (thus they never reach $\langle \text{sink} \rangle$). Diverging runs occur due to non-terminating computations. A terminating run has either terminated successfully, i.e., it passes a \downarrow -state, or it has terminated due to a *violation of an observation*, i.e., it passes the $\langle \downarrow \rangle$ -state. Sets of runs that eventually reach $\langle \downarrow \rangle$, or $\langle \text{sink} \rangle$, or diverge are pairwise disjoint. The \downarrow -labelled states are the *only ones* with positive reward, which is due to the fact that we want to capture probabilities of events (respectively expected values of random variables) occurring at *successful termination* of the program.

The random variables of interest are $\mathbb{E} = \{f \mid f: \mathbb{S} \rightarrow \mathbb{R}_{\geq 0}\}$. Such random variables are referred to as post-expectations [16]. Formally, we have:

Definition 4 (Operational Semantics of Programs). *The operational semantics of a cpGCL program P with respect to a post-expectation $f \in \mathbb{E}$ is the MDP $\mathcal{M}^f[P] = (S, \langle P, \sigma_I \rangle, \text{Act}, \mathcal{P})$ together with a reward function rew , where*

- $S = \{\langle Q, \sigma \rangle, \langle \downarrow, \sigma \rangle \mid Q \text{ is a cpGCL program, } \sigma \in \mathbb{S}\} \cup \{\langle \downarrow \rangle, \langle \text{sink} \rangle\}$ is the countable set of states,
- $\langle P, \sigma_I \rangle \in S$ is the initial state,
- $\text{Act} = \{\text{left}, \text{right}, \text{none}\}$ is the set of actions, and
- \mathcal{P} is the smallest relation defined by the SOS rules given in Figure 1.

The reward function is $\text{rew}(s) = f(\sigma)$ if $s = \langle \downarrow, \sigma \rangle$, and $\text{rew}(s) = 0$, otherwise.

A state of the form $\langle \downarrow, \sigma \rangle$ indicates successful termination, i.e., no commands are left to be executed. These terminal states and the $\langle \downarrow \rangle$ -state go to the $\langle \text{sink} \rangle$ state. **skip** without context terminates successfully. **abort** self-loops, i.e., diverges. $x := E$ alters the variable valuation according to the assignment then terminates successfully. For the concatenation, $\langle \downarrow; Q, \sigma \rangle$ indicates successful termination of the first program, so the execution continues with $\langle Q, \sigma \rangle$. If for $P; Q$ the execution of P leads to $\langle \downarrow \rangle$, $P; Q$ does so, too. Otherwise, for $\langle P, \sigma \rangle \rightarrow \mu$, μ is lifted such that Q is concatenated to the support of μ . For more details on the operational semantics we refer to [4].

If for the conditional choice $\sigma \models G$ holds, P is executed, otherwise Q . The case for **while** is similar. For the probabilistic choice, a distribution ν is created according to probability p . For $\{P\} \square \{Q\}$, we call P the *left* choice and Q the *right* choice for actions $\text{left}, \text{right} \in \text{Act}$. For the **observe** statement, if $\sigma \models G$ then **observe** acts like **skip**. Otherwise, the execution leads directly to $\langle \downarrow \rangle$ indicating a violation of the **observe** statement.

Example 2. Reconsider Example 1, where we set for readability $P_1 = \{x := x + 1\} [0.5] \{c := 1\}$, $P_2 = \text{observe}(\text{“}x \text{ is odd}\text{”})$, $P_3 = \{x := x + 1\}$, and

$$\begin{array}{ll}
\text{(terminal)} \frac{}{\langle \downarrow, \sigma \rangle \longrightarrow \langle \text{sink} \rangle} & \text{(skip)} \frac{}{\langle \text{skip}, \sigma \rangle \longrightarrow \langle \downarrow, \sigma \rangle} \quad \text{(abort)} \frac{}{\langle \text{abort}, \sigma \rangle \longrightarrow \langle \text{abort}, \sigma \rangle} \\
\text{(undesired)} \frac{}{\langle \downarrow \rangle \longrightarrow \langle \text{sink} \rangle} & \text{(assign)} \frac{}{\langle x := E, \sigma \rangle \longrightarrow \langle \downarrow, \sigma[x \leftarrow \llbracket E \rrbracket \sigma] \rangle} \\
\text{(observe1)} \frac{\sigma \models G}{\langle \text{observe } G, \sigma \rangle \longrightarrow \langle \downarrow, \sigma \rangle} & \text{(observe2)} \frac{\sigma \not\models G}{\langle \text{observe } G, \sigma \rangle \longrightarrow \langle \downarrow \rangle} \\
\text{(concatenate1)} \frac{}{\langle \downarrow; Q, \sigma \rangle \longrightarrow \langle Q, \sigma \rangle} & \text{(concatenate2)} \frac{\langle P, \sigma \rangle \longrightarrow \langle \downarrow \rangle}{\langle P; Q, \sigma \rangle \longrightarrow \langle \downarrow \rangle} \\
\text{(concatenate3)} \frac{\langle P, \sigma \rangle \longrightarrow \mu}{\langle P; Q, \sigma \rangle \longrightarrow \nu}, \text{ where } \forall P'. \nu(\langle P'; Q, \sigma' \rangle) := \mu(\langle P', \sigma' \rangle) & \\
\text{(if1)} \frac{\sigma \models G}{\langle \text{ite}(G) \{P\} \{Q\}, \sigma \rangle \longrightarrow \langle P, \sigma \rangle} & \text{(if2)} \frac{\sigma \not\models G}{\langle \text{ite}(G) \{P\} \{Q\}, \sigma \rangle \longrightarrow \langle Q, \sigma \rangle} \\
\text{(while1)} \frac{\sigma \models G}{\langle \text{while}(G) \{P\}, \sigma \rangle \longrightarrow \langle P; \text{while}(G) \{P\}, \sigma \rangle} & \text{(while2)} \frac{\sigma \not\models G}{\langle \text{while}(G) \{P\}, \sigma \rangle \longrightarrow \langle \downarrow, \sigma \rangle} \\
\text{(prob)} \frac{}{\langle \{P\} [p] \{Q\}, \sigma \rangle \longrightarrow \nu}, \text{ where } \nu(\langle P, \sigma \rangle) := p, \nu(\langle Q, \sigma \rangle) := 1 - p & \\
\text{(nondet1)} \frac{}{\langle \{P\} \square \{Q\}, \sigma \rangle \xrightarrow{\text{left}} \langle P, \sigma \rangle} & \text{(nondet2)} \frac{}{\langle \{P\} \square \{Q\}, \sigma \rangle \xrightarrow{\text{right}} \langle Q, \sigma \rangle}
\end{array}$$

Fig. 1. SOS rules for constructing the operational MDP of a cpGCL program. We use $s \longrightarrow t$ to indicate $\mathcal{P}(s, \text{none}, t) = 1$, $s \longrightarrow \mu$ for $\mu \in \text{Distr}(S)$ to indicate $\forall t \in S: \mathcal{P}(s, \text{none}, t) = \mu(t)$, $s \xrightarrow{\text{left}} t$ to indicate $\mathcal{P}(s, \text{left}, t) = 1$, and $s \xrightarrow{\text{right}} t$ to indicate $\mathcal{P}(s, \text{right}, t) = 1$.

$P_4 = \{c := 1\}$. A part of the operational MDP $\mathcal{M}^f \llbracket P \rrbracket$ for an arbitrary initial variable valuation σ_I and post-expectation x is depicted in Figure 2.⁴ Note that this MDP is an MC, as P contains no nondeterministic choices. The MDP has been unrolled until the second loop iteration, i.e., at state $\langle P, \sigma_I[x/2] \rangle$, the unrolling could be continued. The only terminating state is $\langle \downarrow, \sigma_I[x/1, c/1] \rangle$. As our post-expectation is the value of variable x , we assign this value to terminating states, i.e., reward $\boxed{1}$ at state $\langle \downarrow, \sigma_I[x/1, c/1] \rangle$, where x has been assigned 1. At state $\langle P, \sigma_I[c/1] \rangle$, the loop condition is violated as is the subsequent observation because of x being assigned an even number. \triangle

3 Bounded Model Checking for Probabilistic Programs

In this section we describe our approach to model checking probabilistic programs. The key idea is that satisfaction or violation of certain properties for a program can be shown by means of a *finite unrolling* of the program. Therefore, we introduce the notion of a partial operational semantics of a program, which we exploit to apply standard model checking to prove or disprove properties.

First, we state the correspondence between the satisfaction of a property for a cpGCL-program P and for its operational semantics, the MDP $\mathcal{M}^f \llbracket P \rrbracket$. Intu-

⁴ We have tacitly overloaded the variable name x to an expectation here for readability. More formally, by the “expectation x ” we actually mean the expectation $\lambda \sigma. \sigma(x)$.

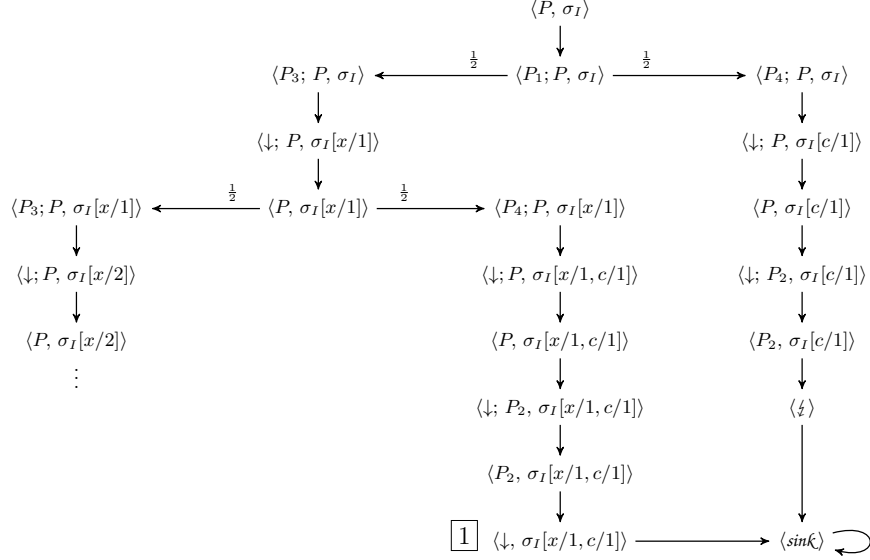


Fig. 2. Partially unrolled operational semantics for program P

itively, a program satisfies a property if and only if the property is satisfied on the operational semantics of the program.

Definition 5 (Satisfaction of Properties). *Given a cpGCL program P and a (conditional) reachability or expected reward property φ . We define*

$$P \models \varphi \quad \text{iff} \quad \mathcal{M}^f[P] \models \varphi .$$

This correspondence on the level of a denotational semantics for cpGCL has been discussed extensively in [17]. Note that there only schedulers which minimize expected rewards were considered. Here, we also need maximal schedulers as we are considering both upper and lower bounds on expected rewards and probabilities. Note that satisfaction of properties is solely based on the operational semantics and induced maximal or minimal probabilities or expected rewards.

We now introduce the notion of a partial operational MDP for a cpGCL-program P , which is a finite approximation of the full operational MDP of P . Intuitively, this amounts to the successive application of SOS rules given in Figure 1, while not all possible rules have been applied yet.

Definition 6 (Partial Operational Semantics). *A partial operational semantics for a cpGCL-program P is a sub-MDP $\mathcal{M}^f[P]' = (S', \langle P, \sigma_I \rangle, Act, \mathcal{P}')$ of the operational semantics for P (denoted $\mathcal{M}^f[P]' \subseteq \mathcal{M}^f[P]$) with $S' \subseteq S$. Let $S_{exp} = S' \setminus \{ \langle Q, \sigma \rangle \in S' \mid Q \neq \downarrow, \exists s \in S \setminus S' \exists \alpha \in Act : \mathcal{P}(\langle Q, \sigma \rangle, \alpha, s) > 0 \}$ be the set of expandable states. Then the transition probability function \mathcal{P}' is*

for $s, s' \in S'$ and $\alpha \in Act$ given by

$$\mathcal{P}'(s, \alpha, s') = \begin{cases} 1, & \text{if } s = s' \text{ for } s, s' \in S_{exp}, \\ \mathcal{P}(s, \alpha, s'), & \text{otherwise.} \end{cases}$$

Intuitively, the set of non-terminating *expandable states* describes the states where there are still SOS rules applicable. Using this definition, the only transitions leaving expandable states are self-loops, enabling to have a well-defined probability measure on partial operational semantics. We will use this for our method, which is based on the fact that both (conditional) reachability probabilities and expected rewards for certain properties will always monotonically increase for further unrollings of a program and the respective partial operational semantics. This is discussed in what follows.

3.1 Growing Expectations

As mentioned before, we are interested in the probability of termination or the expected values of expectations (i.e. random variables ranging over program states) after successful termination of the program. This is measured on the operational MDP by the set of paths *reaching $\langle sink \rangle$ from the initial state conditioned on not reaching $\langle \downarrow \rangle$* [17]. In detail, we have to compute the conditional expected value of post-expectation f after successful termination of program P , given that no observation was violated along the computation. For non-deterministic programs, we have to compute this value either under a minimizing or maximizing scheduler (depending on the given property). We focus our presentation on expected rewards and minimizing schedulers, but all concepts are analogous for the other cases. For $\mathcal{M}^f[P]$ we have

$$\inf_{\mathfrak{S} \in Sched^{\mathcal{M}^f[P]}} ER^{\mathcal{M}^f[P]^{\mathfrak{S}}}(\Diamond \langle sink \rangle \mid \neg \Diamond \langle \downarrow \rangle) .$$

Recall that $\mathcal{M}^f[P]^{\mathfrak{S}}$ is the induced MC under scheduler $\mathfrak{S} \in Sched^{\mathcal{M}^f[P]}$ as in Definition 3. Recall also that for $\neg \Diamond \langle \downarrow \rangle$ all paths not eventually reaching $\langle \downarrow \rangle$ either diverge (collecting reward 0) or pass by a \downarrow -state and reach $\langle sink \rangle$. More importantly, all paths that *do* eventually reach $\langle \downarrow \rangle$ also collect reward 0. Thus:

$$\begin{aligned} & \inf_{\mathfrak{S} \in Sched^{\mathcal{M}^f[P]}} ER^{\mathcal{M}^f[P]^{\mathfrak{S}}}(\Diamond \langle sink \rangle \mid \neg \Diamond \langle \downarrow \rangle) \\ = & \inf_{\mathfrak{S} \in Sched^{\mathcal{M}^f[P]}} \frac{ER^{\mathcal{M}^f[P]^{\mathfrak{S}}}(\Diamond \langle sink \rangle \cap \neg \Diamond \langle \downarrow \rangle)}{Pr^{\mathcal{M}^f[P]^{\mathfrak{S}}}(\neg \Diamond \langle \downarrow \rangle)} \\ = & \inf_{\mathfrak{S} \in Sched^{\mathcal{M}^f[P]}} \frac{ER^{\mathcal{M}^f[P]^{\mathfrak{S}}}(\Diamond \langle sink \rangle)}{Pr^{\mathcal{M}^f[P]^{\mathfrak{S}}}(\neg \Diamond \langle \downarrow \rangle)} . \end{aligned}$$

Finally, observe that the probability of not reaching $\langle \frac{0}{0} \rangle$ is one minus the probability of reaching $\langle \frac{0}{0} \rangle$, which gives us:

$$= \inf_{\mathfrak{S} \in \text{Sched}^{\mathcal{M}^f \llbracket P \rrbracket}} \frac{\text{ER}^{\mathcal{M}^f \llbracket P \rrbracket^{\mathfrak{S}}}(\Diamond \langle \text{sink} \rangle)}{1 - \text{Pr}^{\mathcal{M}^f \llbracket P \rrbracket^{\mathfrak{S}}}(\Diamond \langle \frac{0}{0} \rangle)} . \quad (\dagger)$$

Regarding the quotient minimization we assume “ $\frac{0}{0} < 0$ ” as we see $\frac{0}{0}$ —being undefined—to be less favorable than 0. For programs without nondeterminism this view agrees with a weakest-precondition-style semantics for probabilistic programs with conditioning [17].

It was shown in [18] that *all strict lower bounds* for $\text{ER}^{\mathcal{M}^f \llbracket P \rrbracket^{\mathfrak{S}}}(\Diamond \langle \text{sink} \rangle)$ are in principle computably enumerable in a monotonically non-decreasing fashion. One way to do so, is to allow for the program to be executed for an increasing number of k steps, and collect the expected rewards of all execution traces that have lead to termination within k computation steps. This corresponds naturally to constructing a partial operational semantics $\mathcal{M}^f \llbracket P \rrbracket' \subseteq \mathcal{M}^f \llbracket P \rrbracket$ as in Definition 6 and computing minimal expected rewards on $\mathcal{M}^f \llbracket P \rrbracket'$.

Analogously, it is of course also possible to monotonically enumerate all strict lower bounds of $\text{Pr}^{\mathcal{M}^f \llbracket P \rrbracket^{\mathfrak{S}}}(\Diamond \langle \frac{0}{0} \rangle)$, since—again—we need to just collect the probability mass of all traces that have led to $\langle \frac{0}{0} \rangle$ within k computation steps. Since probabilities are quantities bounded between 0 and 1, a lower bound for $\text{Pr}^{\mathcal{M}^f \llbracket P \rrbracket^{\mathfrak{S}}}(\Diamond \langle \frac{0}{0} \rangle)$ is an upper bound for $1 - \text{Pr}^{\mathcal{M}^f \llbracket P \rrbracket^{\mathfrak{S}}}(\Diamond \langle \frac{0}{0} \rangle)$.

Put together, a lower bound for $\text{ER}^{\mathcal{M}^f \llbracket P \rrbracket^{\mathfrak{S}}}(\Diamond \langle \text{sink} \rangle)$ and a lower bound for $\text{Pr}^{\mathcal{M}^f \llbracket P \rrbracket^{\mathfrak{S}}}(\Diamond \langle \frac{0}{0} \rangle)$ yields a lower bound for (\dagger) . We are thus able to enumerate all lower bounds of $\text{ER}^{\mathcal{M}^f \llbracket P \rrbracket^{\mathfrak{S}}}(\Diamond \langle \text{sink} \rangle \mid \neg \Diamond \langle \frac{0}{0} \rangle)$ by inspection of a finite sub-MDP of $\mathcal{M}^f \llbracket P \rrbracket$. Formally, we have:

Theorem 1. *For a cpGCL program P , post-expectation f , and a partial operational MDP $\mathcal{M}^f \llbracket P \rrbracket' \subseteq \mathcal{M}^f \llbracket P \rrbracket$ it holds that*

$$\begin{aligned} & \inf_{\mathfrak{S} \in \text{Sched}^{\mathcal{M}^f \llbracket P \rrbracket'}} \text{ER}^{\mathcal{M}^f \llbracket P \rrbracket'^{\mathfrak{S}}}(\Diamond \langle \text{sink} \rangle \mid \neg \Diamond \langle \frac{0}{0} \rangle) \\ & \leq \inf_{\mathfrak{S} \in \text{Sched}^{\mathcal{M}^f \llbracket P \rrbracket}} \text{ER}^{\mathcal{M}^f \llbracket P \rrbracket^{\mathfrak{S}}}(\Diamond \langle \text{sink} \rangle \mid \neg \Diamond \langle \frac{0}{0} \rangle) . \end{aligned}$$

3.2 Model Checking

Using Theorem 1, we transfer satisfaction or violation of certain properties from a partial operational semantics $\mathcal{M}^f \llbracket P \rrbracket' \subseteq \mathcal{M}^f \llbracket P \rrbracket$ to the full semantics of the program. For an upper bounded conditional expected reward property $\varphi = \mathbb{E}_{\leq \kappa}(\Diamond T \mid \neg \Diamond U)$ where $T, U \in \mathbb{S}$ we exploit that

$$\mathcal{M}^f \llbracket P \rrbracket' \not\models \varphi \implies P \not\models \varphi . \quad (1)$$

That means, if we can prove the violation of φ on the MDP induced by a finite unrolling of the program, it will hold for all further unrollings, too. This is

because all rewards and probabilities are positive and thus further unrolling can only increase the accumulated reward and/or probability mass.

Dually, for a lower bounded conditional expected reward property $\psi = \mathbb{E}_{\geq \lambda}(\Diamond T \mid \Diamond U)$ we use the following property:

$$\mathcal{M}^f[P]' \models \psi \implies P \models \varphi. \quad (2)$$

The preconditions of Implication (1) and Implication (2) can be checked by probabilistic model checkers like PRISM [5]; this is analogous for conditional reachability properties. Let us illustrate this by means of an example.

Example 3. As mentioned in Example 1, we are interested in the *probability of termination*. As outlined in Section 2.4, this probability can be measured by

$$\Pr(\Diamond \langle \text{sink} \rangle \mid \neg \Diamond \langle \text{!} \rangle) = \frac{\Pr(\Diamond \langle \text{sink} \rangle \wedge \neg \Diamond \langle \text{!} \rangle)}{\Pr(\Diamond \langle \text{!} \rangle)}.$$

We want this probability to be at least $1/2$, i.e., $\varphi = \mathbb{P}_{\geq 0.5}(\Diamond \langle \text{sink} \rangle \mid \neg \Diamond \langle \text{!} \rangle)$. Since for further unrollings of our partially unrolled MDP this probability never decreases, the property can already be verified on the partial MDP $\mathcal{M}^f[P]'$ by

$$\Pr^{\mathcal{M}^f[P]'}(\Diamond \langle \text{sink} \rangle \mid \neg \Diamond \langle \text{!} \rangle) = \frac{1/4}{1/2} = \frac{1}{2},$$

where $\mathcal{M}^f[P]'$ is the sub-MDP from Figure 2. This finite sub-MDP $\mathcal{M}^f[P]'$ is therefore a witness of $\mathcal{M}^f[P] \models \varphi$. \triangle

Algorithmically, this technique relies on suitable heuristics regarding the size of the considered partial MDPs. Basically, in each step k states are expanded and the corresponding MDP is model checked, until either the property can be shown to be satisfied or violated, or no more states are expandable. In addition, heuristics based on shortest path searching algorithms can be employed to favor expandable states that so far induce high probabilities.

Note that this method is a *semi-algorithm* when the model checking problems stated in Implications (1) and (2) are considering strict bounds, i.e. $< \kappa$ and $> \kappa$. It is then guaranteed that the given bounds are finally exceed.

Consider now the case where we want to show *satisfaction* of $\varphi = \mathbb{E}_{\leq \kappa}(\Diamond T \mid \neg \Diamond U)$, i.e., $\mathcal{M}^f[P]' \models \varphi \Rightarrow P \models \varphi$. As the conditional expected reward will monotonically increase as long as the partial MDP is expandable, the implication is only true if there are no more expandable states, i.e., the model is fully expanded. This is analogous for the violation of upper bounded properties. Note that many practical examples actually induce finite operational MDPs which enables to build the full model and perform model checking.

It remains to discuss how this approach can be utilized for parameter synthesis as explained in Section 2.2. For a partial operational pMDP $\mathcal{M}^f[P]'$ and a property $\varphi = \mathbb{E}_{\leq \kappa}(\Diamond T \mid \neg \Diamond U)$ we use tools like PROPhESY [13] to determine for which parameter valuations φ is violated. For each valuation u with $\mathcal{M}^f[P]'[u] \not\models \varphi$ it holds that $\mathcal{M}^f[P][u] \not\models \varphi$; each parameter valuation violating a property on a partial pMDP also violates it on the fully expanded MDP.

4 Evaluation

Experimental Setup. We implemented and evaluated the bounded model checking method in C++. For the model checking functionality, we use the stochastic model checker Storm, developed at RWTH Aachen University, and PROPhESY [19] for parameter synthesis.

We consider five different, well-known benchmark programs, three of which are based on models from the PRISM benchmark suite [5] and others taken from other literature (see Appendix A for some examples). We give the running times of our prototype on several instances of these models. Since there is — to the best of our knowledge — no other tool that can analyze `cpGCL` programs in a purely automated fashion, we cannot meaningfully compare these figures to other tools. As our technique is restricted to establishing that lower bounds on reachability probabilities and the expectations of program variables, respectively, exceed a threshold λ , we need to fix λ for each experiment. For all our experiments, we chose λ to be 90% of the actual value for the corresponding query and choose to expand 10^6 states of the partial operational semantics of a program between each model checking run.

We ran the experiments on an HP BL685C G7 machine with 48 cores clocked with 2.0GHz each and 192GB of RAM while each experiment only runs in a single thread with a time-out of one hour. We ran the following benchmarks⁵:

Crowds Protocol [21]. This protocol aims at anonymizing the sender of R messages by routing them probabilistically through a crowd of N hosts. Some of these hosts, however, are corrupt and try to determine the real sender by observing the host that most recently forwarded a message. For this model, we are interested in (a) the probability that the real sender is observed more than $R/10$ times, and (b) the expected number of times that the real sender is observed.

We also consider a variant (crowds-obs) of the model in which an observe statement ensures that after all messages have been delivered, hosts different from the real sender have been observed at least $R/4$ times. Unlike the model from the PRISM website, our model abstracts from the concrete identity of hosts different from the sender, since they are irrelevant for properties of interest.

Herman Protocol. In this protocol [22], N hosts form a token-passing ring and try to steer the system into a stable state. We consider the probability that the system eventually reaches such a state in two variants of this model where the initial state is either chosen probabilistically or nondeterministically.

Robot. The robot case-study is loosely based on a similar model from the PRISM benchmark suite. It models a robot that navigates through a bounded area of an unbounded grid. Doing so, the robot can be blocked by a janitor that is moving probabilistically across the whole grid. The property of interest is the probability that the robot will eventually reach its final destination.

⁵ All input programs and log files of the experiments can be downloaded at moves.rwth-aachen.de/wp-content/uploads/conference_material/pgcl_atva16.tar.gz

Table 1. Benchmark results for probability queries.

program	instance	#states	#trans.	full?	λ	result	actual	time
crowds	(100,60)	877370	1104290	yes	0.29	0.33	0.33	109
	(100,80)	10^6	1258755	no	0.30	0.33	0.33	131
	(100,100)	$2 \cdot 10^6$	2518395	no	0.30	0.33	0.33	354
crowds-obs	(100,60)	878405	1105325	yes	0.23	0.26	0.26	126
	(100,80)	10^6	1258718	no	0.23	0.25	0.24	170
	(100,100)	$3 \cdot 10^6$	3778192	no	0.23	0.26	0.26	890
herman	(17)	10^6	1136612	no	0.9	0.99	1	91
	(21)	10^6	1222530	no	0.9	0.99	1	142
herman-nd	(13)	1005945	1112188	yes	0.9	1	1	551
	(17)	—	—	no	0.9	0	1	TO
robot	-	181595	234320	yes	0.9	1	1	24
predator	-	10^6	1234854	no	0.9	0.98	1	116
coupon	(5)	10^6	1589528	no	0.75	0.83	0.83	11
	(7)	$2 \cdot 10^6$	3635966	no	0.67	0.72	0.74	440
	(10)	—	—	no	0.57	0	0.63	TO
coupon-obs	(5)	10^6	1750932	no	0.85	0.99	0.99	11
	(7)	10^6	1901206	no	0.88	0.91	0.98	15
	(10)	—	—	no	0.85	0	0.95	TO
coupon-classic	(5)	10^6	1356463	no	3.4e-3	3.8e-3	3.8e-3	9
	(7)	10^6	1428286	no	5.5e-4	6.1e-4	6.1e-4	9
	(10)	—	—	no	3.3e-5	0	3.6e-5	TO

Predator. This model is due to Lotka and Volterra [23, p. 127]. A predator and a prey population evolve with mutual dependency on each other’s numbers. Following some basic biology principles, both populations undergo periodic fluctuations. We are interested in (a) the probability of one of the species going extinct, and (b) the expected size of the prey population after one species has gone extinct.

Coupon Collector. This is a famous example⁶ from textbooks on randomized algorithms [24]. A collector’s goal is to collect all of N distinct coupons. In every round, the collector draws three new coupons chosen uniformly at random out of the N coupons. We consider (a) the probability that the collector possesses all coupons after N rounds, and (b) the expected number of rounds the collector needs until he has all the coupons as properties of interest. Furthermore, we consider two slight variants: in the first one (coupon-obs), an observe statement ensures that the three drawn coupons are all different and in the second one (coupon-classic), the collector may only draw one coupon in each round.

Table 1 shows the results for the probability queries. For each model instance, we give the number of explored states and transitions and whether or not the model was fully expanded. Note that the state number is a multiple of 10^6 in

⁶ https://en.wikipedia.org/wiki/Coupon_collector%27s_problem

Table 2. Benchmark results for expectation queries.

program	instance	#states	#trans.	full?	result	actual	time
crowds	(100,60)	877370	1104290	yes	5.61	5.61	125
	(100,80)	10^6	1258605	no	7.27	7.47	176
	(100,100)	$2 \cdot 10^6$	2518270	no	9.22	9.34	383
crowds-obs	(100,60)	878405	1105325	yes	5.18	5.18	134
	(100,80)	10^6	1258569	no	6.42	6.98	206
	(100,100)	$2 \cdot 10^6$	2518220	no	8.39	8.79	462
predator	—	$3 \cdot 10^6$	3716578	no	99.14	?	369
coupon	(5)	10^6	1589528	no	4.13	4.13	15
	(7)	$3 \cdot 10^6$	5379492	no	5.86	6.38	46
	(10)	—	—	no	0	10.1	TO
coupon-obs	(5)	10^6	1750932	no	2.57	2.57	13
	(7)	$2 \cdot 10^6$	3752912	no	4.22	4.23	30
	(10)	—	—	no	0	6.96	TO
coupon-classic	(5)	10^6	1356463	no	11.41	11.42	15
	(7)	10^6	1393360	no	18.15	18.15	21
	(10)	—	—	no	0	29.29	TO

case the model was not fully explored, because our prototype always expands 10^6 states before it does the next model checking call. The next three columns show the probability bound (λ), the result that the tool could achieve as well as the actual answer to the query on the full (potentially infinite) model. Due to space constraints, we rounded these figures to two significant digits. We report on the time in seconds that the prototype took to establish the result (TO = 3600 sec.).

We observe that for most examples it suffices to perform few unfolding steps to achieve more than 90% of the actual probability. For example, for the largest crowds-obs program, $3 \cdot 10^6$ states are expanded, meaning that three unfolding steps were performed. Answering queries on programs including an observe statement can be costlier (crowds vs. crowds-obs), but does not need to be (coupon vs. coupon-obs). In the latter case, the observe statement prunes some paths early that were not promising to begin with, whereas in the former case, the observe statement only happens at the very end, which intuitively makes it harder for the search to find target states. We are able to obtain non-trivial lower bounds for all but two case studies. For herman-nd, not all of the (nondeterministically chosen) initial states were explored, because our exploration order currently does not favour states that influence the obtained result the most. Similarly, for the largest coupon collector examples, the time limit did not allow for finding one target state. Again, an exploration heuristic that is more directed towards these could potentially improve performance drastically.

Table 2 shows the results for computing the expected value of program variables at terminating states. For technical reasons, our prototype currently cannot perform more than one unfolding step for this type of query. To achieve mean-

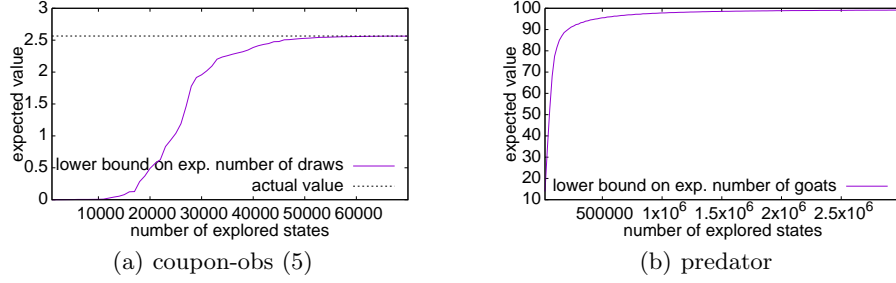


Fig. 3. The obtained values approach the actual value from below.

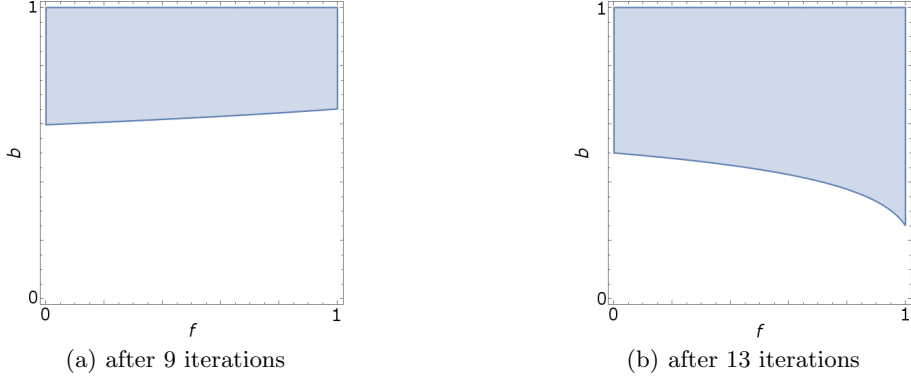


Fig. 4. Analyzing parametric models yields violating parameter instances.

ingful results, we therefore vary the number of explored states until 90% of the actual result is achieved. Note that for the predator program, the actual value for the query is not known to us, so we report on the value at which the result only grows very slowly. The results are similar to the probability case in that most often a low number of states suffices to show meaningful lower bounds. Unfortunately — as before — we can only prove a trivial lower bound for the largest coupon collector examples.

Figure 3 illustrates how the obtained lower bounds approach the actual expected value with increasing number of explored states for two case studies. For example, in the left picture one can observe that exploring 60000 states is enough to obtain a very precise lower bound on the expected number of rounds the collector needs to gather all five coupons, as indicated by the dashed line.

Finally, we analyze a parametric version of the crowds model that uses the parameters f and b to leave the probabilities (i) for a crowd member to be corrupt (b) and (ii) of forwarding (instead of delivering) a message (f) unspecified. In each iteration of our algorithm, we obtain a rational function describing a lower bound on the actual probability of observing the real sender of the message

more than once *for each parameter valuation*. Figure 4 shows the regions of the parameter space in which the protocol was determined to be unsafe (after iterations 9 and 13, respectively) in the sense that the probability to identify the real sender exceeds $\frac{1}{2}$. Since the results obtained over different iterations are monotonically increasing, we can conclude that all parameter valuations that were proved to be unsafe in some iteration are in fact unsafe in the full model. This in turn means that the blue area in Figure 4 grows in each iteration.

5 Conclusion and Future Work

We presented a direct verification method for probabilistic programs employing probabilistic model checking. We conjecture that the basic idea would smoothly translate to reasoning about recursive probabilistic programs [25]. In the future we are interested in how loop invariants [26] can be utilized to devise complete model checking procedures preventing possibly infinite loop unrollings. This is especially interesting for reasoning about covariances [27], where a mixture of invariant-reasoning and successively constructing the operational MC would yield sound over- and underapproximations of covariances. To extend the gain for the user, we will combine this approach with methods for counterexamples [28], which can be given in terms of the programming language [29,19]. Moreover, it seem promising to investigate how approaches to automatically *repair* a probabilistic model towards satisfaction of properties [30,31] can be transferred to programs.

References

1. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: FOSE, ACM Press (2014) 167–181
2. Sankaranarayanan, S., Chakarov, A., Gulwani, S.: Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In: PLDI, ACM (2013) 447–458
3. Claret, G., Rajamani, S.K., Nori, A.V., Gordon, A.D., Borgström, J.: Bayesian inference using data flow analysis. In: ESEC/SIGSOFT FSE, ACM Press (2013) 92–102
4. Gretz, F., Katoen, J.P., McIver, A.: Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. *Perform. Eval.* **73** (2014) 110–132
5. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV. Volume 6806 of LNCS, Springer (2011) 585–591
6. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: IscasMC: A web-based probabilistic model checker. In: FM. Volume 8442 of LNCS, Springer (2014) 312–317
7. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. *Performance Evaluation* **68**(2) (2011) 90–104
8. Kattenbelt, M.: Automated Quantitative Software Verification. PhD thesis, Oxford University (2011)

9. Sharir, M., Pnueli, A., Hart, S.: Verification of probabilistic programs. *SIAM Journal on Computing* **13**(2) (1984) 292–314
10. Vardi, M.Y.: Automatic verification of probabilistic concurrent finite-state programs. In: *FOCS*, IEEE Computer Society (1985) 327–338
11. Baier, C., Katoen, J.P.: *Principles of Model Checking*. The MIT Press (2008)
12. Baier, C., Klein, J., Klüppelholz, S., Märcker, S.: Computing conditional probabilities in Markovian models efficiently. In: *TACAS*. Volume 8413 of *LNCS*, Springer (2014) 515–530
13. Dehnert, C., Junges, S., Jansen, N., Corzilius, F., Volk, M., Bruintjes, H., Katoen, J., Ábrahám, E.: Prophesy: A probabilistic parameter synthesis tool. In: *CAV*. Volume 9206 of *LNCS*, Springer (2015) 214–231
14. Quatmann, T., Dehnert, C., Jansen, N., Junges, S., Katoen, J.: Parameter synthesis for Markov models: Faster than ever. *CoRR* **abs/1602.05113** (2016)
15. Dijkstra, E.W.: *A Discipline of Programming*. Prentice Hall (1976)
16. McIver, A., Morgan, C.: *Abstraction, Refinement And Proof For Probabilistic Systems*. Springer (2004)
17. Jansen, N., Kaminski, B.L., Katoen, J., Olmedo, F., Gretz, F., McIver, A.: Conditioning in probabilistic programming. *Electr. Notes Theor. Comput. Sci.* **319** (2015) 199–216
18. Kaminski, B.L., Katoen, J.P.: On the hardness of almost-sure termination. In: *MFCS*. Volume 9234 of *LNCS*, Springer (2015)
19. Dehnert, C., Jansen, N., Wimmer, R., Ábrahám, E., Katoen, J.: Fast debugging of PRISM models. In: *ATVA*. Volume 8837 of *LNCS*, Springer (2014) 146–162
20. Jansen, N., Dehnert, C., Kaminski, B.L., Katoen, J., Westhofen, L.: Bounded model checking for probabilistic programs. *CoRR* **abs/1605.04477** (2016)
21. Reiter, M.K., Rubin, A.D.: Crowds: Anonymity for web transactions. *ACM Trans. on Information and System Security* **1**(1) (1998) 66–92
22. Herman, T.: Probabilistic self-stabilization. *Inf. Process. Lett.* **35**(2) (1990) 63–67
23. Brauer, F., Castillo-Chavez, C.: *Mathematical Models in Population Biology and Epidemiology*. Texts in Applied Mathematics. Springer New York (2001)
24. Erdős, P., Rnyi, A.: On a classical problem of probability theory. *Publ. Math. Inst. Hung. Acad. Sci., Ser. A* **6** (1961) 215 – 220
25. Olmedo, F., Kaminski, B., Katoen, J.P., Matheja, C.: Reasoning about recursive probabilistic programs. In: *LICS*. (2016) [to appear].
26. Gretz, F., Katoen, J.P., McIver, A.: PRINSYS - on a quest for probabilistic loop invariants. In: *QEST*. Volume 8054 of *LNCS*, Springer (2013) 193–208
27. Kaminski, B., Katoen, J.P., Matheja, C.: Inferring covariances for probabilistic programs. In: *QEST*. Volume 9826 of *LNCS*, Springer (2016) [to appear].
28. Ábrahám, E., Becker, B., Dehnert, C., Jansen, N., Katoen, J., Wimmer, R.: Counterexample generation for discrete-time Markov models: An introductory survey. In: *SFM*. Volume 8483 of *Lecture Notes in Computer Science*, Springer (2014) 65–121
29. Wimmer, R., Jansen, N., Abraham, E., Katoen, J.P.: High-level counterexamples for probabilistic automata. *Logical Methods in Computer Science* **11**(1:15) (2015)
30. Bartocci, E., Grosu, R., Katsaros, P., Ramakrishnan, C.R., Smolka, S.A.: Model repair for probabilistic systems. In: *TACAS*. Volume 6605 of *Lecture Notes in Computer Science*, Springer (2011) 326–340
31. Pathak, S., Ábrahám, E., Jansen, N., Tacchella, A., Katoen, J.P.: A greedy approach for the efficient repair of stochastic models. In: *NFM*. Volume 9058 of *LNCS*, Springer (2015) 295–309

A Models

A.1 coupon-obs (5)

```
1 int coup0 := 0;
2 int coup1 := 0;
3 int coup2 := 0;
4 int coup3 := 0;
5 int coup4 := 0;
6
7 int draw1 := 0;
8 int draw2 := 0;
9 int draw3 := 0;
10
11 int numberDraws := 0;
12
13 while (!(coup0 = 1) | !(coup1 = 1) | !(coup2 = 1) | !(coup3 =
    1) | !(coup4 = 1)) {
14   draw1 := unif(0,4);
15   draw2 := unif(0,4);
16   draw3 := unif(0,4);
17   numberDraws := numberDraws + 1;
18
19   observe (draw1 != draw2 & draw1 != draw3 & draw2 != draw3);
20
21   if(draw1 = 0 | draw2 = 0 | draw3 = 0) {
22     coup0 := 1;
23   }
24   if(draw1 = 1 | draw2 = 1 | draw3 = 1) {
25     coup1 := 1;
26   }
27   if(draw1 = 2 | draw2 = 2 | draw3 = 2) {
28     coup2 := 1;
29   }
30   if (draw1 = 3 | draw2 = 3 | draw3 = 3) {
31     coup3 := 1;
32   }
33   if (draw1 = 4 | draw2 = 4 | draw3 = 4) {
34     coup4 := 1;
35   }
36 }
```

A.2 coupon (5)

```
1 int coup0 := 0;
2 int coup1 := 0;
3 int coup2 := 0;
4 int coup3 := 0;
5 int coup4 := 0;
6
7 int draw1 := 0;
8 int draw2 := 0;
9 int draw3 := 0;
10
11 int numberDraws := 0;
12
13 while (!(coup0 = 1) | !(coup1 = 1) | !(coup2 = 1) | !(coup3 =
    1) | !(coup4 = 1)) {
14   draw1 := unif(0,4);
15   draw2 := unif(0,4);
16   draw3 := unif(0,4);
17   numberDraws := numberDraws + 1;
18
19   if(draw1 = 0 | draw2 = 0 | draw3 = 0) {
20     coup0 := 1;
21   }
22   if(draw1 = 1 | draw2 = 1 | draw3 = 1) {
23     coup1 := 1;
24   }
25   if(draw1 = 2 | draw2 = 2 | draw3 = 2) {
26     coup2 := 1;
27   }
28   if (draw1 = 3 | draw2 = 3 | draw3 = 3) {
29     coup3 := 1;
30   }
31   if (draw1 = 4 | draw2 = 4 | draw3 = 4) {
32     coup4 := 1;
33   }
34 }
```

A.3 crowds-obs (100, 60)

```
1 int delivered := 0;
2 int lastSender := 0;
3 int remainingRuns := 60;
4 int observeSender := 0;
5 int observeOther := 0;
6
7 while(remainingRuns > 0) {
8   while(delivered = 0) {
9     {
10      if(lastSender = 0) {
11        observeSender := observeSender + 1;
12      } else {
13        observeOther := observeOther + 1;
14      }
15      lastSender := 0;
16      delivered := 1;
17    } [0.091] {
18      {
19        { lastSender:=0; } [1/100] { lastSender := 1; }
20      }
21      [0.8]
22      {
23        lastSender := 0;
24        // When not forwarding, the message is delivered here
25        delivered := 1;
26      }
27    }
28  }
29  // Set up new run.
30  delivered := 0;
31  remainingRuns := remainingRuns - 1;
32 }
33 observe(observeOther > 15);
```

A.4 crowds (100, 60)

```
1 int delivered := 0;
2 int lastSender := 0;
3 int remainingRuns := 60;
4 int observeSender := 0;
5 int observeOther := 0;
6
7 while(remainingRuns > 0) {
8   while(delivered = 0) {
9     {
10      if(lastSender = 0) {
11        observeSender := observeSender + 1;
12      } else {
13        observeOther := observeOther + 1;
14      }
15      lastSender := 0;
16      delivered := 1;
17    } [0.091] {
18      {
19        { lastSender:=0; } [1/100] { lastSender := 1; }
20      }
21      [0.8]
22      {
23        lastSender := 0;
24        // When not forwarding, the message is delivered here
25        delivered := 1;
26      }
27    }
28  }
29  // Set up new run.
30  delivered := 0;
31  remainingRuns := remainingRuns - 1;
32 }
```

A.5 crowds (100, 60) parametric

This program is parametric with the parameters f (probability of forwarding the message) and b (probability that a crowd member is bad).

```
1 int delivered := 0;
2 int lastSender := 0;
3 int remainingRuns := 60;
4 int observeSender := 0;
5 int observeOther := 0;
6
7 while(remainingRuns > 0) {
8   while(delivered = 0) {
9     {
10      if(lastSender = 0) {
11        observeSender := observeSender + 1;
12      } else {
13        observeOther := observeOther + 1;
14      }
15      lastSender := 0;
16      delivered := 1;
17    } [b] {
18      {
19        { lastSender:=0; } [1/100] { lastSender := 1; }
20      }
21      [f]
22      {
23        lastSender := 0;
24        // When not forwarding, the message is delivered here
25        delivered := 1;
26      }
27    }
28  }
29  // Set up new run.
30  delivered := 0;
31  remainingRuns := remainingRuns - 1;
32 }
```